A High-Level Semantics for Program Execution under Total Store Order Memory

Brijesh Dongol[1,*], Oleg Travkin[2], John Derrick[1,*], and Heike Wehrheim[2]

[1] Department of Computer Science
The University of Sheffield, S1 4DP, UK
[2] Fakultät für Elektrotechnik, Informatik und Mathematik
The University of Paderborn, Germany
B.Dongol@sheffield.ac.uk, oleg82@zitmail.uni-paderborn.de,
J.Derrick@dcs.shef.ac.uk, wehrheim@mail.uni-paderborn.de

Abstract. Processor cores within modern multicore systems often communicate via shared memory and use (local) store buffers to improve performance. A penalty for this improvement is the loss of Sequential Consistency to weaker memory guarantees that increase the number of possible program behaviours, and hence, require a greater amount of programming effort. This paper formalises the effect of Total Store Order (TSO) memory — a weak memory model that allows a write followed by a read in the program order to be reordered during execution. Although the precise effects of TSO are well-known, a high-level formalisation of programs that execute under TSO has not been developed. We present an interval-based semantics for programs that execute under TSO memory and include methods for fine-grained expression evaluation, capturing the non-determinism of both concurrency and TSO-related reorderings.

1 Introduction

Approaches to reasoning about concurrency usually assume *Sequentially Consistent* (SC) memory models, where program instructions are executed by the hardware in the order specified by the program [19], i.e., under SC memory, execution of the sequential composition $S_1\ ;\ S_2$ of statements S_1 and S_2 must execute S_2 after S_1. Fig. 1 shows a multicore architecture idealised by the SC memory model, where processor cores interact directly with shared memory. In such an architecture, contention for shared memory becomes a bottleneck to efficiency, and hence, modern processors often utilise additional local buffers within which data may be stored (e.g., the processor cores in Fig. 2 use local write buffers). Furthermore, modern processors implement weaker memory models than sequential consistency and allow the order in which instructions are executed to differ from the program order in a restricted manner [1], e.g., $\mathit{Write} \to \mathit{Read}$, $\mathit{Write} \to \mathit{Write}$, $\mathit{Read} \to \mathit{Write}$, $\mathit{Read} \to \mathit{Read}$. Here $\mathit{Write} \to \mathit{Read}$ means that a *Write* instruction to an address a followed by a *Read* instruction to an address b in the program order are allowed to be reordered if $a \neq b$. As a result, a programmer must perform additional reasoning to ensure that the actual (executed) behaviour of a program is consistent with the expected behaviour.

[*] This research is supported by EPSRC Grant EP/J003727/1.

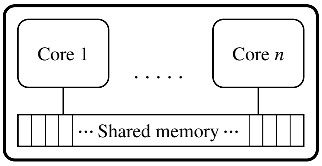


Fig. 1. Idealised multicore architecture

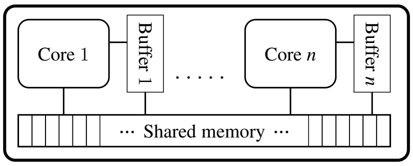


Fig. 2. Multicore architecture with write buffers

In this paper, we study the high-level behaviour of the common x86 multicore processor architecture. Each core uses a write buffer (as shown in Fig. 2), which is a FIFO queue that stores pending writes. A processor core performing a write may enqueue the write in the buffer and continue computation without waiting for the write to be committed to memory. Pending writes do not become visible to other cores until the buffer is flushed, which commits (some or all) pending writes. Thus, x86 architectures allow *Write* $\rightarrow$ *Read* reordering. Furthermore, using a technique known as *Intra-Process Forwarding* (IPF) [16] a processor core may read pending writes from its own (local) write buffer, i.e., without accessing shared memory. The combination of *Write* $\rightarrow$ *Read* reordering and IPF forms the *Total Store Order* (TSO) memory model [1, 24].

Existing approaches to memory-model-aware reasoning, e.g. Alglave et al [2], formalise several different orders that are imposed by a specific memory model. Applying these orders to a program yields all possible behaviour that can be observed with respect to the applied memory model. Executable memory models like the x86-TSO [23, 24] have been defined to observe the impact of a memory model on a program's execution. Such models can be used for state space exploration, but this quickly becomes infeasible due to the exponential explosion in the complexity of the state space. Burckhardt et al use an approach [6] in which the memory model is defined axiomatically and combined with a set of axioms modelling a program written in a low-level language. The combination of both is used to feed a SAT-solver to check for program properties like linearisability [15]. Each of the approaches [2, 6, 23, 24] is focused on the use of a low-level language instead of the high-level language in which programs are often written. Hence, to perform a verification, programs need to be observed and understood in their low-level representation, which is a complex task because at this level of abstraction, programs are verbose in their representation and use additional variables to implement high-level language instructions.

Although there are many approaches dealing with the influence of memory models for low-level languages [2–4, 23, 24], we are not aware of any approach that tries to lift such memory model effects to a higher level of abstraction. Our work here is hence unique in this sense. The basic idea is to think of high-level statements as being executed over an interval of time or an execution window. Such execution windows can overlap, if programs are executed concurrently. Under TSO memory, the execution windows can even overlap within a single process. Overlapping windows correspond to program instructions that can be executed in any order, representing the effect of concurrent executions and reorderings due to TSO. Furthermore, overlapping execution windows may also interfere with each other and fixing the outcome of an execution within a window can influence the outcome within another.

Initially: $x = 0 \wedge y = 0 \wedge z \neq 0$

Process p	Process q
p_1: $x := 1$;	q_1: $y := 1$;
p_2: **if** $y = 0 \wedge z = 0$	q_2: $z := x$
p_3: **then** *statement*$_1$	
p_4: **else** *statement*$_2$	

Fig. 3. SC does not allow execution of *statement*$_1$, TSO does

Initially: $x = 0 \wedge y = 0 \wedge z \neq 0$

Process p	Process q
p_1: $\mathit{write}(x, 1)$;	q_1: $\mathit{write}(y, 1)$;
$p_{2.1}$: $\big(\mathit{read}(y, r1_p)$;	$q_{2.1}$: $\mathit{read}(x, r_q)$;
$p_{2.2}$: $\mathit{read}(z, r2_p)\big)$	$q_{2.2}$: $\mathit{write}(z, r_q)$;
$\sqcap$	
$p_{2.3}$: $\big(\mathit{read}(z, r2_p)$;	
$p_{2.4}$: $\mathit{read}(y, r1_p)\big)$;	
$p_{2.5}$: **if** $r1_p = 0 \wedge r2_p = 0 \ldots$	

Fig. 4. Low-level representation of program in Fig. 3

Section 2 introduces the TSO memory model and its influence on a program's behaviour. Section 3 presents our interval-based framework for reasoning about different memory models, an abstract programming language, and a parameterised semantics for the language. In Section 4, we formalise instantaneous and actual states evaluation under SC memory, a restricted form of TSO that allows *Write* $\dashrightarrow$ *Read* reordering without allowing IPF, and *Write* $\dashrightarrow$ *Read* with IPF to fully cover TSO behaviour.

2 Effect of Total Store Order on Program Behaviour

On top of the non-determinism inherent within concurrent programs, TSO memory allows additional relaxations that enable further reordering of program instructions within a process via *Write* $\dashrightarrow$ *Read* reordering and IPF, complicating their analysis [4]. We describe these concepts and their effects on program behaviour using the examples in Sections 2.1 and 2.2. Note that *Write* $\dashrightarrow$ *Read* reordering is not implemented without IPF by any current processor, but we find it useful to consider its effects separately.

2.1 *Write* $\dashrightarrow$ *Read* Reordering

Fig. 3 shows a program with two concurrent processes p and q that use shared variables x, y and z. A low-level representation of Fig. 3 is given in Fig. 4, which uses additional local registers $r1_p$, $r2_p$ and r_q.[1] Evaluation of the guard at p_2 is split into a number of atomic steps, where the order in which y and z are read is chosen non-deterministically. That is, after execution of p_1, either $p_{2.1}$; $p_{2.2}$ or $p_{2.3}$; $p_{2.4}$ is executed. For both choices, under SC memory, process p will never execute *statement*$_1$ because whenever control of process p is at $p_{2.5}$, either $r1_p$ or $r2_p$ is non-zero, and hence, the guard at $p_{2.5}$ always evaluates to *false*. In particular, for SC memory, if $r1_p = 0$ holds, then either $p_{2.1}$ or $p_{2.4}$ must have been executed before q_1 (otherwise $r1_p$ would equal 1), and hence, by the program order (which is preserved by the execution order), p_1 must have been executed before q_1. Thus, if $r1_p = 0$ holds, then $r2_p \neq 0$ must hold, and hence, the guard at

[1] Note that implementation of the **if** statement in process p uses additional local variables and goto/jump instructions, whose details have been elided.

$p_{2.5}$ must evaluate to *false*. Furthermore, if $r2_p = 0$ holds at $p_{2.5}$, then $q_{2.2}$ must have been executed before p_1 (otherwise z with value 1 would be loaded as the value of r_2). Therefore, due to the program order, q_1 must also have been executed before p_1, and hence, before both $p_{2.1}$ and $p_{2.4}$. However, this means $r1_p = 1$ must hold at $p_{2.5}$.

Now consider a *restricted TSO* (RTSO) memory model that allows *Write* $\to$ *Read* reordering but without IPF. For example, RTSO allows $p_{2.1}$ in Fig. 4 to be executed before p_1 even though p_1 occurs before $p_{2.1}$ in the program order. All other program orders are preserved, including a write to a variable followed by a read to the same variable. Execution of the program in Fig. 4 under RTSO allows execution of $statement_1$ if process p chooses branch $p_{2.1}$; $p_{2.2}$ (i.e., p reads y then z) to evaluate the guard at p_2. This occurs if both:

1. p_1 ; $p_{2.1}$; $p_{2.2}$; $p_{2.5}$ is reordered to $p_{2.1}$; p_1 ; $p_{2.2}$; $p_{2.5}$, which can happen if the write to x (i.e., instruction p_1) is stored in p's write buffer, but committed to memory before execution of $p_{2.2}$, and
2. q_1 ; $q_{2.1}$; $q_{2.2}$ is reordered to $q_{2.1}$; q_1 ; $q_{2.2}$, which can happen if the write to y (i.e., q_1) is stored in q's write buffer.

After the reordering, the concurrent execution of p and q may execute $p_{2.1}$ (setting $r1_p = 0$), then $q_{2.1}$; q_1 ; $q_{2.2}$ (setting $z = 0$), and then $p_{2.2}$ (setting $r2_p = 0$).

Note that it is also possible for none of the instructions to be re-ordered, in which case execution under RTSO would be identical to execution under SC memory. Furthermore, if process p chooses branch $p_{2.3}$; $p_{2.4}$, $statement_1$ cannot be executed despite any reorderings within p and q. Finally, RTSO does not allow re-orderings such as $p_{2.2}$; $p_{2.1}$ because they are both read instructions (i.e., *Read* $\to$ *Read* ordering is preserved), $q_{2.2}$; q_1 because both $q_{2.2}$ and q_1 are write instructions (i.e., *Write* $\to$ *Write* ordering is preserved), and $q_{2.2}$; $q_{2.1}$ because $q_{2.1}$ is a read and $q_{2.2}$ is a write (i.e., *Read* $\to$ *Write* ordering is preserved). A write to a variable that is followed by a read to the same variable in the program order must not be reordered (e.g., in Fig. 6, reordering $p_{2.1}$; p_1 is disallowed).

2.2 Total Store Order

TSO extends RTSO by including IPF, allowing a process to read pending writes from its own buffer, and hence, obtaining values that are not yet globally visible to other processes. To observe the effect of IPF, consider the program in Fig. 5 and its corresponding low-level representation in Fig. 6. Process p can never execute $statement_1$ under RTSO memory because the read at $p_{2.1}$ cannot be reordered with the write at p_1 due to the variable dependency. Furthermore, because *Read* $\to$ *Read* ordering is preserved, $p_{2.1}$ prevents reads to y at $p_{3.1}$ and $p_{3.4}$ from being executed before the write instruction at p_1 even though both reorderings $p_{3.1}$; $p_{2.2}$ and $p_{3.3}$; $p_{3.4}$; $p_{2.2}$ are possible. Similarly, $q_{2.1}$ prevents $q_{3.1}$ from being executed before q_1 even though $q_{2.2}$ may be reordered with $q_{3.1}$. Because SC memory is a special case of RTSO in which no reorderings are possible, it is also not possible for p to reach $statement_1$ under SC memory.

In contrast, TSO allows execution of $statement_1$ because IPF enables reads to occur from the write buffer. For the program in Fig. 6, the value written by $write(x, 1)$ at p_1 could still be in p's write buffer, which could be used by $p_{2.1}$ before the write at p_1 is committed to memory. Then $write(u, r0_p)$ at $p_{2.2}$ may become a pending write, and

Initially: $x = 0 \wedge y = 0 \wedge z \neq 0$

Process p	Process q
p_1: $x := 1$;	q_1: $y := 1$;
p_2: $u := x$;	q_2: $v := y$;
p_3: **if** $y = 0 \wedge z = 0$	q_3: $z := x$
p_4: **then** *statement*$_1$;	
p_5: **else** *statement*$_2$	

Fig. 5. Neither SC nor RTSO cause execution of *statement*$_1$, TSO does

Initially: $x = 0 \wedge y = 0 \wedge z \neq 0$

Process p	Process q
p_1: $write(x, 1)$;	q_1: $write(y, 1)$;
$p_{2.1}$: $read(x, r0_p)$;	$q_{2.1}$: $read(y, r0_q)$;
$p_{2.2}$: $write(u, r0_p)$;	$q_{2.2}$: $write(v, r0_q)$;
$p_{3.1}$: $\big(read(y, r1_p)$;	$q_{3.1}$: $read(x, r1_q)$;
$p_{3.2}$: $read(z, r2_p)\big)$	$q_{3.2}$: $write(z, r1_q)$
$\sqcap$	
$p_{3.3}$: $\big(read(z, r2_p)$;	
$p_{3.4}$: $read(y, r1_p)\big)$;	
$p_{3.5}$: **if** $r1_p = 0 \wedge r2_p = 0 \ldots$	

Fig. 6. Low-level representation of program in Fig. 5

then $read(y, r1_p)$ and $read(z, r2_p)$ (at $p_{3.1}$ and $p_{3.2}$, respectively) may be executed. By fetching values from memory before the pending $write(x, 1)$ at p_1 has been committed, the reads at $p_{3.1}$ and $p_{3.2}$, can appear as if they were executed before $p_{2.1}$. The same arguments apply to process q where $read(y, r0_q)$ at $q_{2.1}$ can read the value of y from q's write buffer, and hence, execution of $q_{2.1}$ and $q_{3.1}$ appear to be reordered. A concurrent execution after reordering that allows control to reach p_4 is:

$$p_{3.1} \,;\, q_{3.1} \,;\, q_1 \,;\, q_{2.1} \,;\, q_{2.2} \,;\, q_{3.2} \,;\, p_1 \,;\, p_{2.1} \,;\, p_{2.2} \,;\, p_{3.2}$$

This example shows that TSO allows $Read \rightarrow Read$ reordering in a restricted manner and in fact that the IPF relaxation can be viewed as such [3].

3 Interval-Based Reasoning

The programs in Figs. 4 and 6 have helped explain TSO concepts, however, reasoning about interleavings at such a low level of abstraction quickly becomes infeasible. Instead, we use a framework that considers the intervals in which a program execute [9], which enables both non-deterministic evaluation [13] and compositional reasoning [17]. We present interval predicates in Section 3.1, fractional permissions (to model conflicting accesses) in Section 3.2, and a programming language as well as its generalised interval-based semantics in Section 3.3.

3.1 Interval Predicates

We use interval predicates to formalise the interval-based semantics due to the generality they provide over frameworks that consider programs as relations between pre/post states. An *interval* is a contiguous set of integers (denoted $\mathbb{Z}$), and hence the set of all intervals is $Intv \mathrel{\widehat{=}} \{\Delta \subseteq \mathbb{Z} \mid \forall t_1, t_2 : \Delta \bullet \forall t : \mathbb{Z} \bullet t_1 \leq t \leq t_2 \Rightarrow t \in \Delta\}$. Using '.' for function application (i.e., $f.x$ denotes $f(x)$), we let $\mathsf{lub}.\Delta$ and $\mathsf{glb}.\Delta$ denote the *least upper* and *greatest lower* bounds of an interval Δ, respectively. We define $\mathsf{lub}.\varnothing \mathrel{\widehat{=}} -\infty$, $\mathsf{glb}.\varnothing \mathrel{\widehat{=}} \infty$, $\mathsf{inf}.\Delta \mathrel{\widehat{=}} (\mathsf{lub}.\Delta = \infty)$, $\mathsf{fin}.\Delta \mathrel{\widehat{=}} \neg\mathsf{inf}.\Delta$, and $\mathsf{empty}.\Delta \mathrel{\widehat{=}} (\Delta = \varnothing)$.

One must often reason about two *adjoining* intervals, i.e., intervals that immediately precede or follow a given interval. For $\Delta_1, \Delta_2 \in \mathit{Intv}$, we say Δ_1 *adjoins* Δ_2 iff $\Delta_1 \propto \Delta_2$ holds, where $\Delta_1 \propto \Delta_2 \mathrel{\widehat{=}} (\Delta_1 \cup \Delta_2 \in \mathit{Intv}) \wedge (\forall t_1 \colon \Delta_1, t_2 \colon \Delta_2 \bullet t_1 < t_2)$. Thus, $\Delta_1 \propto \Delta_2$ holds iff Δ_2 immediately follows Δ_1. Note that adjoining intervals Δ_1 and Δ_2 must be both contiguous and disjoint, and that both $\Delta \propto \varnothing$ and $\varnothing \propto \Delta$ trivially hold.

Given that variable names are taken from the set *Var*, a *state space* over a set of variables $V \subseteq \mathit{Var}$ is given by $\mathit{State}_V \mathrel{\widehat{=}} V \to \mathit{Val}$ and a *state* is a member of State_V, i.e., a state is a total function mapping variables in V to values in *Val*. A *stream* of behaviours over V is given by the total function $\mathit{Stream}_V \mathrel{\widehat{=}} \mathbb{Z} \to \mathit{State}_V$, which maps each time in $\mathbb{Z}$ to a state over V. A *predicate* over type T is a total function $\mathcal{P}T \mathrel{\widehat{=}} T \to \mathbb{B}$ mapping each member of T to a Boolean. For example $\mathcal{P}\mathit{State}_V$ and $\mathcal{P}\mathit{Stream}_V$ denote state and stream predicates, respectively. To facilitate reasoning about specific parts of a stream, we use *interval predicates*, which have type $\mathit{IntvPred}_V \mathrel{\widehat{=}} \mathit{Intv} \to \mathcal{P}\mathit{Stream}_V$. A stream predicate defines the behaviour of a system over all time, and an interval predicate defines the behaviour of a system with respect to a given interval [9, 10]. We assume pointwise lifting of operators on stream and interval predicates in the normal manner, e.g., if g_1 and g_2 are interval predicates, Δ is an interval and s is a stream, we have $(g_1 \wedge g_2).\Delta.s = (g_1.\Delta.s \wedge g_2.\Delta.s)$.

We define two operators on interval predicates: *chop* (to model sequential composition), and *k*- and *ω-iteration* (to model loops), i.e.,

$$(g_1 \ ; \ g_2).\Delta.s \mathrel{\widehat{=}} \begin{pmatrix} \exists \Delta_1, \Delta_2 \colon \mathit{Intv} \bullet (\Delta = \Delta_1 \cup \Delta_2) \wedge \\ \qquad (\Delta_1 \propto \Delta_2) \wedge g_1.\Delta_1.s \wedge g_2.\Delta_2.s \end{pmatrix} \vee (\mathsf{inf}.\Delta \wedge g_1.\Delta.s)$$

$$g^0 \mathrel{\widehat{=}} \mathsf{empty} \qquad g^{k+1} \mathrel{\widehat{=}} g^k \ ; \ g \qquad g^\omega \mathrel{\widehat{=}} \nu z \bullet (g \ ; \ z) \vee \mathsf{empty}$$

The *chop* operator ';' is a basic operator on two interval predicates [21, 10], where $(g_1 \ ; \ g_2).\Delta.s$ holds iff either interval Δ may be split into two adjoining parts Δ_1 and Δ_2 so that g_1 holds for Δ_1 and g_2 holds for Δ_2 in s, or the least upper bound of Δ is ∞ and g_1 holds for Δ in s. Inclusion of the second disjunct $\mathsf{inf}.\Delta \wedge g_1.\Delta.s$ enables g_1 to model an infinite (divergent or non-terminating) program. Iteration g^k defines the k-fold iteration of g and g^ω is the greatest fixed point of $\lambda z \bullet (g \ ; \ z) \vee \mathsf{empty}$, which allows both finite and infinite iterations of g [12]. We use

$$(\ominus g).\Delta.s \mathrel{\widehat{=}} \exists \Omega \colon \mathit{Intv} \bullet \Omega \propto \Delta \wedge g.\Omega.s$$

to denote that g holds in some interval Ω that immediately precedes Δ.

We define the following operators to formalise properties over an interval using a state predicate c over an interval Δ in stream s.

$$(\boxdot c).\Delta.s \mathrel{\widehat{=}} \forall t \colon \Delta \bullet c.(s.t) \qquad (\diamondsuit c).\Delta.s \mathrel{\widehat{=}} \exists t \colon \Delta \bullet c.(s.t)$$

$$\overrightarrow{c}.\Delta.s \mathrel{\widehat{=}} (\mathsf{lub}.\Delta \in \Delta) \wedge c.(s.(\mathsf{lub}.\Delta))$$

That is $(\boxdot c).\Delta.s$ holds iff c holds for each state $s.t$ where $t \in \Delta$, $(\diamondsuit c).\Delta.s$ holds iff c holds in some state $s.t$ where $t \in \Delta$, and $\overrightarrow{c}.\Delta.s$ holds iff c holds in the state corresponding to the end of Δ. Note that $\boxdot c$ trivially holds for an empty interval, but $\diamondsuit c$ and $\overrightarrow{c}$ do not. A variable v is stable over interval Δ in stream s iff $\mathsf{stable}.v.\Delta.s$ holds, where $\mathsf{stable}.v.\Delta.s \mathrel{\widehat{=}} \exists k \colon \mathit{Val} \bullet \ominus(\overrightarrow{v = k}) \wedge \boxdot(v = k)$.

3.2 Fractional Permissions

The behaviour of a process executing a command is formalised by an interval predicate, and the behaviour of a parallel execution over an interval is given by the conjunction of these behaviours over the same interval. Because the state-spaces of the two processes often overlap, there is a possibility that a process writing to a variable conflicts with a read or write to the same variable by another process. To ensure that such conflicts do not take place, we follow Boyland's idea of mapping variables to a *fractional permission* [5], which is *rational* number between 0 and 1. A process has write-only access to a variable v if its permission to access v is 1, has read-only access to v if its permission to access v is above 0 but below 1, and has no access to v if its permission to access v is 0. Note that a process may not have both read and write permission to a variable. Because a permission is a rational number, read access to a variable may be split arbitrarily (including infinitely) among the processes of the system. However, at most one process may have write permission to a variable in any given state.

We assume that every state contains a *permission* variable Π whose value in state $\sigma \in State_V$ is a function of type $V \to Proc \to \{n{:}\, \mathbb{Q} \mid 0 \le n \le 1\}$, where *Proc* denotes the type of a process identifier. Note that it is possible for permissions to be distributed differently within states σ_1, σ_2 even if the values of the normal variables in σ_1 and σ_2 are identical. Process $p \in Proc$ has *write-permission* to variable v in state σ iff $\mathscr{W}_p.v.\sigma \mathrel{\widehat{=}} (\sigma.\Pi.v.p = 1)$, has *read-permission* to v in σ iff $\mathscr{R}_p.v.\sigma \mathrel{\widehat{=}} (0 < \sigma.\Pi.v.p < 1)$, and has *no-permission* to access v in σ iff $\mathscr{D}_p.v.\sigma \mathrel{\widehat{=}} (\sigma.\Pi.v.p = 0)$ holds. In the context of a stream s, for any time $t \in \mathbb{Z}$, process p may only write to and read from v in the transition step from $s.(t-1)$ to $s.t$ if $\mathscr{W}_p.v.(s.t)$ and $\mathscr{R}_p.v.(s.t)$ hold, respectively. Thus, $\mathscr{W}_p.v.(s.t)$ does not grant process p permission to write to v in the transition from $s.t$ to $s.(t+1)$ (and similarly $\mathscr{R}_p.v.(s.t)$). We introduce two assumptions on streams using fractional permissions that formalise our assumptions on the underlying hardware.

HC1. If no process has write access to v within an interval, then the value of v does not change within the interval, i.e., for any interval Δ and stream s, $(\boxdot(\forall p{:}\, Proc \bullet \neg \mathscr{W}_p.v) \Rightarrow \mathsf{stable}.v)\,.\Delta.s$

HC2. The sum of the permissions of the processes on any variable v is at most 1, i.e., for any interval Δ and stream s, $(\boxdot((\Sigma_{p \in Proc} \Pi.v.p) \le 1)).\Delta.s$

For the rest of this paper, we assume that the streams and intervals under consideration satisfy both **HC1** and **HC2**. Further restrictions may explicitly be introduced to the programs if required. In essence, both **HC1** and **HC2** are implicit *rely* conditions of the programs that we develop [9, 17].

3.3 A Programming Language

To formalise common programming constructs, we present a language inspired by the refinement calculus [20], extended to enable reasoning about concurrency. The syntax closely matches program code, which simplifies translation from an implementation to the model. For a state predicate b, variable v, expression e and set of processes $P \subseteq Proc$, the abstract syntax of commands is given by *Cmd* below, where $BC \in BasicCmd$ and $C, C_1, C_2, C_p \in Cmd$.

$$\begin{aligned}
BasicCmd &::= \mathsf{Idle} \mid [b] \mid v := e \\
Cmd &::= BC \mid \mathsf{Empty} \mid \mathsf{Magic} \mid \mathsf{Chaos} \mid \mathsf{fin\_Idle} \mid \mathsf{inf\_Idle} \mid \\
&\quad C_1 \,;\, C_2 \mid C_1 \sqcap C_2 \mid C^{\omega} \mid \|_{p:P}\, C_p \mid \textsc{Init}\, b \bullet C
\end{aligned}$$

Thus, a basic command may either be Idle, a guard $[b]$ or an assignment $v := e$. A command may either be a basic command, Empty (representing the empty program), Magic (an infeasible command that has no behaviours), Chaos (a chaotic command that allows any behaviour), fin_Idle (a finite idle), inf_Idle (an infinite idle), sequential composition $(C_1 \,;\, C_2)$, non-deterministic choice $C_1 \sqcap C_2$, iteration C^{ω}, parallel composition $\|_{p:P}\, C_p$, or a command with an initialisation $\textsc{Init}\, b \bullet C$.

Using this syntax, the programs in Fig. 3 and Fig. 5 are modelled by the commands in Fig. 7 and Fig. 8, respectively, where the labels in Figs. 3 and 5 have been omitted. For Fig. 3, the initialisation is modelled using the $\textsc{Init}$ construct, and the main command consists of the parallel composition between C_p and C_q, which model processes p and q, respectively. Command C_p is the sequential composition of the assignment followed by a non deterministic choice between Ct_p and Cf_p, which respectively model the true and false evaluations of the guard at p_2 in Fig. 3.

We define an interval-based semantics for this language, which is used to formalise program execution in RTSO ($\mathcal{R}$) and TSO ($\mathcal{T}$) memory models. Like [9], we split SC executions into instantaneous ($\mathcal{I}$) and apparent states ($\mathcal{S}$) evaluation, where the apparent states stem from non-atomic expression evaluation, i.e., by observing different variables of an expression at different times [9, 13].

To simplify comparison of the different memory models on program execution, we present a generalised semantics where the behaviour function is parameterised by the memory model under consideration. In particular, the generalised semantics for commands in a memory model $\mathcal{M} \in \{\mathcal{I}, \mathcal{S}, \mathcal{R}, \mathcal{T}\}$ is given by function $[\![\cdot]\!]_P^{\mathcal{M}}$ in Fig. 9, which for a given command returns an interval predicate that formalises the behaviour of the command with respect to $P \subseteq Proc$. A basic command BC is assumed to be executed by a single process p and its behaviour over an interval with respect to memory model $\mathcal{M}$ is defined by $(\!| BC |\!)_p^{\mathcal{M}}$, which requires that we instantiate interval predicates $\mathsf{idle}_p^{\mathcal{M}}$, $\mathsf{eval}_p^{\mathcal{M}}$ and $\mathsf{update}_p^{\mathcal{M}}$. Note that the behaviour of an assignment consists of two portions, an evaluation portion, where the expression e is evaluated to some value k, followed by an interval in which the variable v is updated to a new value k.

Note that the behaviours of each of the commands except for basic commands and parallel composition decompose for each of the memory models in the same way. The behaviour of Empty, Magic and Chaos are always empty, *false* and *true*, respectively, sequential composition is defined by the chop operator, and non-deterministic choice is defined by disjunction. The behaviour of command iteration C^{ω} is defined as iteration of

$$\begin{aligned}
Ct_p &\mathrel{\widehat{=}} [y = 0 \wedge z = 0] \,;\, statement_1 \\
Cf_p &\mathrel{\widehat{=}} [y \neq 0 \vee z \neq 0] \,;\, statement_2 \\
C_p &\mathrel{\widehat{=}} x := 1 \,;\, (Ct_p \sqcap Cf_p) \\
C_q &\mathrel{\widehat{=}} y := 1 \,;\, z := x \\
C &\mathrel{\widehat{=}} \textsc{Init}\, x = 0 \wedge y = 0 \wedge z \neq 0 \bullet C_p \| C_q
\end{aligned}$$

Fig. 7. Formalisation of program in Fig. 3

$$\begin{aligned}
Dt_p &\mathrel{\widehat{=}} [y = 0 \wedge z = 0] \,;\, statement_1 \\
Df_p &\mathrel{\widehat{=}} [y \neq 0 \vee z \neq 0] \,;\, statement_2 \\
D_p &\mathrel{\widehat{=}} x := 1 \,;\, u := x \,;\, (Dt_p \sqcap Df_p) \\
D_q &\mathrel{\widehat{=}} y := 1 \,;\, v := y \,;\, z := x \\
D &\mathrel{\widehat{=}} \textsc{Init}\, x = 0 \wedge y = 0 \wedge z \neq 0 \bullet D_p \,\|\, D_q
\end{aligned}$$

Fig. 8. Formalisation of program in Fig. 5

$$(\!|\,\mathsf{Idle}\,|\!)_p^{\mathcal{M}} \mathrel{\widehat{=}} \mathsf{idle}_p^{\mathcal{M}}.Var \qquad (\!|\,[b]\,|\!)_p^{\mathcal{M}} \mathrel{\widehat{=}} \mathsf{eval}_p^{\mathcal{M}}.b \qquad (\!|\,v := e\,|\!)_p^{\mathcal{M}} \mathrel{\widehat{=}} \exists k{:}\,Val \bullet \mathsf{eval}_p^{\mathcal{M}}.(e = k)\ ;\ \mathsf{update}_p^{\mathcal{M}}(v,k)$$

$$[\![\,BC\,]\!]_{\{p\}}^{\mathcal{N}} \mathrel{\widehat{=}} (\!|BC|\!)_p^{\mathcal{N}} \qquad [\![\,\mathsf{Magic}\,]\!]_P^{\mathcal{M}} \mathrel{\widehat{=}} false \qquad [\![\,C_1\ ;\ C_2\,]\!]_P^{\mathcal{M}} \mathrel{\widehat{=}} [\![\,C_1\,]\!]_P^{\mathcal{M}}\ ;\ [\![\,C_2\,]\!]_P^{\mathcal{M}}$$

$$[\![\,\mathsf{Empty}\,]\!]_P^{\mathcal{M}} \mathrel{\widehat{=}} \mathsf{empty} \qquad [\![\,\mathsf{Chaos}\,]\!]_P^{\mathcal{M}} \mathrel{\widehat{=}} true \qquad [\![\,C_1 \sqcap C_2\,]\!]_P^{\mathcal{M}} \mathrel{\widehat{=}} [\![\,C_1\,]\!]_P^{\mathcal{M}} \vee [\![\,C_2\,]\!]_P^{\mathcal{M}}$$

$$[\![\,C^{\omega}\,]\!]_P^{\mathcal{M}} \mathrel{\widehat{=}} ([\![\,C\,]\!]_P^{\mathcal{M}})^{\omega}$$

$$[\![\,\mathsf{fin\_Idle}\,]\!]_P^{\mathcal{M}} \mathrel{\widehat{=}} \mathsf{fin} \wedge \textstyle\bigwedge_{p:P}[\![\,\mathsf{Idle}\,]\!]_p^{\mathcal{M}} \qquad [\![\,\mathsf{inf\_Idle}\,]\!]_P^{\mathcal{M}} \mathrel{\widehat{=}} \mathsf{inf} \wedge \textstyle\bigwedge_{p:P}[\![\,\mathsf{Idle}\,]\!]_p^{\mathcal{M}}$$

$$[\![\,\textsc{Init}\,b \bullet C\,]\!]_P^{\mathcal{M}} \mathrel{\widehat{=}} \ominus\overrightarrow{b} \Rightarrow [\![\,C\,]\!]_P^{\mathcal{M}}$$

$$term.S.T \mathrel{\widehat{=}} S \in \{\mathsf{fin\_Idle}, \mathsf{inf\_Idle}\} \wedge T \in \{\mathsf{fin\_Idle}, \mathsf{inf\_Idle}\} \wedge (S = \mathsf{inf\_Idle} \Rightarrow T \neq \mathsf{inf\_Idle})$$

$$\begin{aligned}[\![\,\|_{p:P}\,C_p\,]\!]_P^{\mathcal{N}} \mathrel{\widehat{=}}\ & \textbf{if}\ P = \varnothing\ \textbf{then}\ true\ \textbf{elseif}\ P = \{p\}\ \textbf{then}\ [\![\,C_p\,]\!]_{\{p\}}^{\mathcal{M}} \\ & \textbf{else}\ \exists Q,R,S,T \bullet (Q \cup R = P) \wedge (Q \cap R = \varnothing) \wedge Q \neq \varnothing \wedge R \neq \varnothing \wedge \\ & \qquad term.S.T \wedge [\![\,(\|_{p:Q}\,C_p)\ ;\ S\,]\!]_Q^{\mathcal{M}} \wedge [\![\,(\|_{p:R}\,C_p)\ ;\ T\,]\!]_R^{\mathcal{M}}\end{aligned}$$

Fig. 9. General semantics for interval-based reasoning

the behaviour of C, and the behaviour of the $\textsc{Init}\,b \bullet C$ is the behaviour of C assuming that b holds at the end of some immediately preceding interval.

Assuming $\mathcal{N} \mathrel{\widehat{=}} \mathcal{M}\backslash\{\mathcal{T}\}$, the behaviour of a basic command $[\![\,BC\,]\!]_{\{p\}}^{\mathcal{N}}$ is defined as the basic behaviour $(\!|BC|\!)_{\{p\}}^{\mathcal{N}}$. Behaviour $[\![\,\|_{p:P}\,C_p\,]\!]_P^{\mathcal{N}}$ is *true* if the set P is empty and discards the parallel composition operator if P is a singleton set. If P contains at least two elements, $[\![\,\|_{p:P}\,C_p\,]\!]_P^{\mathcal{N}}$ holds if P can be split into two non-empty disjoint subsets Q and R such that both $[\![\,\|_{p:Q}\,C_p\ ;\ S\,]\!]_Q^{\mathcal{N}}$ and $[\![\,\|_{p:R}\,C_p\ ;\ T\,]\!]_R^{\mathcal{N}}$ hold, where S and T denote possible idling. This idling is necessary because $\|_{p:Q}\,C_p$ and $\|_{p:R}\,C_p$ may terminate at different times [9] and idling may sometimes be infinite because a component may not terminate. Within a parallel composition, fractional permissions together with assumptions **HC1** and **HC2**, restrict access to shared variables, and hence, how processes may affect each other [9]. The behaviours of both $[\![\,BC\,]\!]_{\{p\}}^{\mathcal{T}}$ and $[\![\,\|_{p:P}\,C_p\,]\!]_P^{\mathcal{T}}$ (i.e., for TSO memory) are defined in Section 4.4.

4 Program Semantics under Different Memory Models

We present a semantics for instantaneous evaluation (where an entire expression is evaluated in a single atomic step) in Section 4.1 and apparent states evaluation (where variables are assumed to be read one at a time) is given in Section 4.2. This work has appeared in [9], but we present it here once again for completeness and to simplify comparisons with RTSO (Section 4.3) and TSO memory (Section 4.4).

4.1 Sequentially Consistent Instantaneous Evaluation Semantics

The simplest execution model we consider is $\mathcal{I}$ (instantaneous evaluation), where expressions are evaluated under SC in one of the actual states that occur in an interval of evaluation [13]. Given that an expression e is evaluated in an interval $\varDelta$ of stream s and that S is the set of states of s that occur within $\varDelta$, this form of expression evaluation returns a value of e for some state of S. To formalise this, we define interval predicate

$$\mathsf{idle}_p.V \mathrel{\widehat{=}} \forall v{:}\, V \bullet \boxdot\neg\mathscr{W}_p.v$$

i.e., p does not write to any variable of V. To complete the instantaneous evaluation semantics for our language, we instantiate interval predicates $\mathsf{idle}_p^{\mathcal{I}}$, $\mathsf{eval}_p^{\mathcal{I}}$ and $\mathsf{update}_p^{\mathcal{I}}$ as follows, where c is a state predicate, v is a variable and k is a value. We let $vars.c$ denote the set of free variables of c.

$$\mathsf{idle}_p^{\mathcal{I}} \mathrel{\widehat{=}} \mathsf{idle}_p \qquad \mathsf{eval}_p^{\mathcal{I}}.c \mathrel{\widehat{=}} \Diamond(c \wedge (\forall v{:}\, vars.c \bullet \mathscr{R}_p.v)) \wedge \mathsf{idle}_p.Var$$
$$\mathsf{update}_p^{\mathcal{I}}(v,k) \mathrel{\widehat{=}} \boxdot((v = k) \wedge \mathscr{W}_p.v) \wedge \neg\mathsf{empty} \wedge \mathsf{idle}_p.(Var\backslash\{v\})$$

The semantics of $\mathsf{idle}_p^{\mathcal{I}}$ is straightforward. Evaluation of c in a stream s within interval Δ is given by $\mathsf{eval}_p^{\mathcal{I}}.c.\Delta.s$, which holds iff (a) there is a time $t \in \Delta$ such that $c.(s.t)$ holds and p has permission to read the variables of c in $s.t$, and (b) p does not write to any variable within Δ. Updating the value of v to k (in shared memory) within interval Δ of stream s is modelled by $\mathsf{update}_p^{\mathcal{I}}(v,k).\Delta.s$, which holds iff (a) throughout Δ, v has value k and p has write permission to v, (b) Δ is non-empty and (c) p does not write to any other variable. We must ensure that $\neg\mathsf{empty}$ holds because $\boxdot c$ is trivially true for an empty interval.

4.2 Sequentially Consistent Apparent States Evaluation Semantics

Instantaneous evaluation is not problematic for expressions in which at most one variable of the expression is unstable [9, 13]. For more complex expressions (e.g., the guard of p_2 in Fig. 3), instantaneous evaluation will be unimplementable because hardware will seldom be able to guarantee that all variables of an expression can be read in a single atomic step. That is, instantaneous evaluation does not reflect the fact that implementations can read at most one variable atomically. Hence, we consider a second method of evaluation that returns a value in the states apparent to a process.

For each expression evaluation, we assume that each variable is read at most once, and that the same value is used for each occurrence of the variable in an expression[2]. We assume that a compiler non-deterministically chooses an ordering of read instructions when evaluating an expression. For example, in the low-level program in Fig. 4, the order of reads of the variables of p_2 in Fig. 3 is non-deterministically chosen.

The $apparent_{p,W}^{\mathcal{S}}$ function generates a set of states that may not exist in the stream, but can be observed by a process that reads variables one at a time. For example, eliding details of the permission variable, if over an interval Δ, a stream s has actual states $\{x \mapsto 0, y \mapsto 0\}$, $\{x \mapsto 1, y \mapsto 0\}$, $\{x \mapsto 1, y \mapsto 1\}$, a possible observable state within Δ in s is $\{x \mapsto 0, y \mapsto 1\}$. To generate the set of states apparent to process p, one must ensure that p has the appropriate read permissions. Using the $apparent_{p,W}^{\mathcal{S}}$ function, we define the *possibly* operator $\circledS_p$, which evaluates state predicates over a set of apparent states with respect to a given interval and stream.

$$apparent_{p,W}^{\mathcal{S}}.\Delta.s \mathrel{\widehat{=}} \{\sigma{:}\, State_W \mid \forall\, v{:}\, W \bullet \exists t{:}\, \Delta \bullet (\sigma.v = s.t.v) \wedge \mathscr{R}_p.v.(s.t)\}$$
$$(\circledS_p c).\Delta.s \mathrel{\widehat{=}} \exists \sigma{:}\, apparent_{p,vars.c}^{\mathcal{S}}.\Delta.s \bullet c.\sigma$$

[2] It is possible to define evaluators that, for example, (re)read a variable for each occurrence of the variable, and hence, potentially returns false for $v = v$ if the value of v changes during the observation interval [18, 13].

To complete the program semantics for sequentially consistent apparent states evaluation, we must instantiate predicates $\mathsf{idle}_p^{\mathcal{S}}$, $\mathsf{eval}_p^{\mathcal{S}}$ and $\mathsf{update}_p^{\mathcal{S}}$ for a process p.

$$\mathsf{idle}_p^{\mathcal{S}} \mathrel{\widehat{=}} \mathsf{idle}_p \qquad \mathsf{eval}_p^{\mathcal{S}}.c \mathrel{\widehat{=}} (\circledS_p c) \wedge \mathsf{idle}_p^{\mathcal{I}}.\mathit{Var} \qquad \mathsf{update}_p^{\mathcal{S}}(v,k) \mathrel{\widehat{=}} \mathsf{update}_p^{\mathcal{I}}(v,k)$$

Except for $\mathsf{eval}_p^{\mathcal{S}}.c$, these interval predicates are identical to memory model $\mathcal{I}$. Interval predicate $\mathsf{eval}_p^{\mathcal{S}}.c$ uses $\circledS$ evaluation, which models the fact that the variables of c are read one at a time and at most once in the interval of evaluation, capturing the non-determinism due to fine-grained concurrency, e.g., Fig. 4. We ask the reader to consult [9, 13] for further details on instantaneous and apparent states evaluation under SC memory.

4.3 Restricted TSO

As described in Section 2.1, RTSO weakens SC memory by relaxing *Write* $\to$ *Read* ordering, but a read from a variable with a pending write must wait for the pending write to be committed to memory. RTSO is not implemented by any hardware, however, we use it as a stepping stone to formalisation of the more complicated TSO model in Section 4.4, which is implemented by several mainstream processors [7, 16, 22].

As with apparent states evaluation in Section 4.2, the semantics of a program under RTSO is defined by instantiating interval predicates $\mathsf{idle}_p^{\mathcal{R}}$, $\mathsf{eval}_p^{\mathcal{R}}$ and $\mathsf{update}_p^{\mathcal{R}}$ for a process p, which requires that we formalise expression evaluation with respect to the reorderings RTSO memory may cause. Like SC apparent states evaluation (Section 4.2), we assume that the variables of an expression may be read in any order, but that each variable is read at most once per evaluation. We define an apparent states evaluator $\mathit{apparent}_{p,W}^{\mathcal{R}}.\Delta.s$, where Δ is the interval of execution in the *program order* in stream s. Because SC is not guaranteed, the interval in which an expression is evaluated (i.e., the *execution order*) extends beyond Δ (see Fig. 10). We use *write/read barrier* variables $\mathit{WB}_p \notin \mathit{Var}$ and $\mathit{RB}_p \notin \mathit{Var}$ for each process p, which describe how far the interval of evaluation may extend. By selecting placement of the barriers, one can control the reorderings allowed by RTSO. We assume that a write barrier for each variable is placed at initialisation, and hence, a variable's value prior to initialisation cannot be read.

Like permission variable Π, we implicitly assume that each state of the program includes barrier variables WB_p and RB_p for each process p. A write barrier for variable v in process p prevents reorderings of reads to variable v within p, and hence, its value is a function of type $\mathit{Var} \to \mathbb{B}$. Process p places a write barrier to a variable v whenever the value of v is updated, i.e., committed to memory (see definition of $\mathsf{update}_p^{\mathcal{R}}$ below). This prevents future reads to v in process p from being reordered with the write to v.

Read barriers must allow variables that are part of the same expression to be read in any order, but must disallow reorderings from expressions that are evaluated later in the program order. Hence, one is required to uniquely identify each expression occurrence. The value of a read RB_p variable is hence of type $\mathbb{Z} \to \mathit{Var} \to \mathbb{B}$, where the integer component is used to identify the corresponding expression evaluation. In particular, we identify an evaluation using the least upper bound of the interval of evaluation. Hence, whenever a process p reads a variable v in an interval Δ as part of an expression evaluation, p places a read barrier for v with identifier $\mathsf{lub}.\Delta$ at the time at which v is

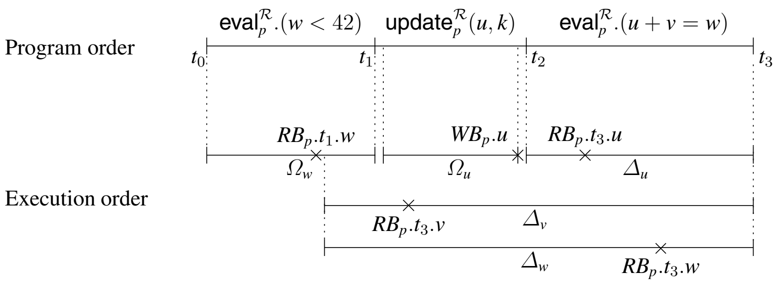


Fig. 10. Extending apparent states evaluation

read. This prevents any reads that are part of future expression evaluations from being reordered with the read to v in Δ, and hence, from reading outdated values. We define apparent states evaluation for RTSO as follows, where $id \in \mathbb{Z}$, $v \in Var$, $p \in Proc$, $s \in Stream$, $W \subseteq Var$ and $\Delta \in Intv$.

$$\begin{array}{l} extendedIntv.id.v.p.s \mathrel{\widehat{=}} \\ \left\{ \Omega\colon Intv \;\middle|\; \begin{array}{l} (\mathsf{lub}.\Omega = id) \wedge \\ \boxdot(\neg WB_p.v \wedge (\forall t\colon \mathbb{Z}, u\colon Var \bullet RB_p.t.u \Rightarrow t \geq id)).\Omega.s \end{array} \right\} \end{array}$$

$$\begin{array}{l} apparent^{\mathcal{R}}_{p,W}.\Delta.s \mathrel{\widehat{=}} \\ \left\{ \sigma\colon State \;\middle|\; \begin{array}{l} \forall v\colon W \bullet \exists \Omega\colon extendedIntv.(\mathsf{lub}.\Delta).v.p.s \bullet \exists t\colon \Omega \bullet \\ \qquad (\sigma.v = s.t.v) \wedge \mathscr{R}_p.v.(s.t) \wedge RB_p.(\mathsf{lub}.\Delta).v.(s.t) \end{array} \right\} \end{array}$$

Hence, $extendedIntv.id.v.p.s$ returns a set of extended intervals within which p may read v with respect to stream s as part of the expression identified by id. Each interval within $extendedIntv.id.v.p.s$ must not contain a write barrier to v or a read barrier to any variable with identifier t such that $t < id$. An example of such extended intervals is given in Fig. 10, where intervals Δ_v and Δ_w (corresponding to the evaluation of $u + v = w$) are disallowed from extending beyond the read barrier $RB_p.t_1.w$, which marks the point at which w was read in Ω_w. Interval Δ_u is disallowed from extending beyond time t_2, due to the write barrier for u ($WB.u$) within Ω_u that is placed by the update to u. The write barrier $WB.u$ in Ω_u does not affect Δ_v and Δ_w because $u \neq v$ and $u \neq w$, and hence, allows v and w to be evaluated before u is updated. The read barriers in Δ_u, Δ_v and Δ_w do not affect each other, because they each have the same identifier t_3, i.e., are part of the same expression evaluation. However, note that any evaluations that occur after t_3 in the program order would be disallowed from extending beyond the latest read barrier identified by t_3, which in the example above is $RB_p.t_3.w$ within Δ_w.

The apparent states for RTSO are defined by $apparent^{\mathcal{R}}_{p,W}$, where extended intervals are used for evaluation of each variable. To generate a state σ apparent to process p, for each variable $v \in W$, we pick an extended interval Ω corresponding to v, then pick a time t from Ω such that p has permission to read v at t, and set the value of v in σ to $(s.t).v$. Process p places a read barrier to v with identifier $\mathsf{lub}.\Delta$ at t to prevent future reads to any variable in the program order from being reordered with the read to v at time t in the execution order.

Using the set of apparent states, we define $(\Diamond\!\!\!\!\circledR_p c).\varDelta.s$ which holds iff state predicate c holds in some state apparent to process p in interval $\varDelta$ and stream s with respect to RTSO memory.

$$(\Diamond\!\!\!\!\circledR_p c).\varDelta.s \mathrel{\widehat{=}} \exists \sigma\colon apparent^{\mathcal{R}}_{p,vars.c}.\varDelta.s \bullet c.\sigma$$

In addition to the effect of each command on the read/write permissions, we must also specify the effect of each command on the read/write barriers. We define the following interval predicates for a process p, set of variables V, interval $\varDelta$ and stream s.

$$\begin{aligned}\mathsf{wBar}_p.V.\varDelta.s &\mathrel{\widehat{=}} \forall v\colon V \bullet \boxdot\neg WB_p.v.\varDelta.s\\ \mathsf{rBar}_p.V.\varDelta.s &\mathrel{\widehat{=}} \forall v\colon V \bullet \boxdot\neg RB_p.(\mathsf{lub}.\varDelta).v.\varDelta.s\end{aligned}$$

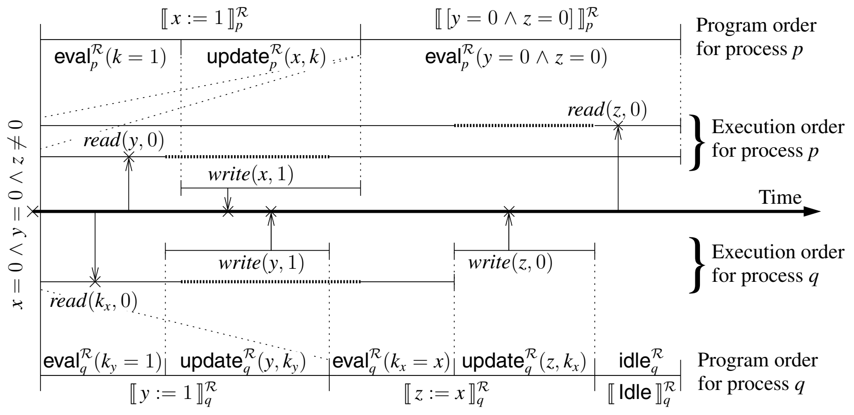
Hence, $\mathsf{wBar}_p.V.\varDelta.s$ states that p does not place any write barriers to any of the variables of V within $\varDelta$ and $\mathsf{rBar}_p.V.\varDelta.s$ states that p does not place any read barrier to any of the variables of V with identifier $\mathsf{lub}.\varDelta$ within $\varDelta$.

This now allows one to complete the semantics of programs that execute under RTSO memory, which is achieved by the following instantiations:

$$\begin{aligned}\mathsf{idle}^{\mathcal{R}}_p.V &\mathrel{\widehat{=}} (\mathsf{idle}_p \wedge \mathsf{rBar}_p \wedge \mathsf{wBar}_p).V\\ \mathsf{eval}^{\mathcal{R}}_p.c &\mathrel{\widehat{=}} \Diamond\!\!\!\!\circledR_p c \wedge (\mathsf{idle}_p \wedge \mathsf{wBar}_p).Var \wedge \mathsf{rBar}_p.(Var\backslash vars.c)\\ \mathsf{update}^{\mathcal{R}}_p(v,k) &\mathrel{\widehat{=}} \mathsf{update}^{\mathcal{I}}_p(v,k) \wedge \overrightarrow{WB_p.v} \wedge \mathsf{wBar}_p.(Var\backslash\{v\}) \wedge \mathsf{rBar}_p.Var\end{aligned}$$

Hence, $\mathsf{idle}^{\mathcal{R}}_p.\varDelta.s$ holds iff p does not write to any variable and does not introduce any read/write barriers. Interval predicate $\mathsf{eval}^{\mathcal{R}}_p.c.\varDelta.s$ holds iff c holds in some apparent state generated by $apparent^{\mathcal{R}}_{p,vars.c}.\varDelta.s$, process p does not write to any variable, and introduces no barriers except for the read barriers for variables used in c. Finally, a variable update to v behaves in the same manner as $\mathsf{update}^{\mathcal{I}}_p(v,k)$ and additionally places a write barrier to v at the end of execution. An update does not introduce any other barriers except for the one to v.

Example. We apply our RTSO semantics to our running example program from Fig. 3 using the encoding from Fig. 7. Instead of unrolling the full details of our definitions, we consider Fig. 11, which shows a possible interval of execution of processes $C_p \parallel C_q$ that leads to execution of $statement_1$. Note that details regarding disjointness at the boundary between adjoining intervals have been elided from the diagram. The top of Fig. 11 shows process p and its corresponding basic commands, obtained by unfolding the language definitions in Fig. 9. Below this, we present the actual intervals of execution allowed by the weak memory model; corresponding intervals of the program and execution orders are connected by dotted lines. Representation of process q is vertically inverted. The time line shows the times at which the actual reads/writes of each basic command occur in terms of the low-level instructions from Fig. 4. The intervals in which the updates occur in both p and q are preserved by the execution order. However, the intervals in which $\mathsf{eval}^{\mathcal{T}}_p(y = 0 \wedge z = 0)$ and $\mathsf{eval}^{\mathcal{T}}_q(k_x = x)$ (which is part of the behaviour of $z := x$) execute extend beyond their respective intervals in the program order. As a result of the extension, process p may read $y = 0$, process q may write $z = 0$, which allows process p to read $z = 0$. Note that the intervals in which the reads occur also contain a fuzzy portion depicting an interval in which read permission is not available due to a write in the other process. Furthermore, our framework allows



$[\![x := 1]\!]_p^{\mathcal{R}}$
$[\![[y = 0 \wedge z = 0]]\!]_p^{\mathcal{R}}$
Program order for process p
$\mathsf{eval}_p^{\mathcal{R}}(k = 1)$
$\mathsf{update}_p^{\mathcal{R}}(x, k)$
$\mathsf{eval}_p^{\mathcal{R}}(y = 0 \wedge z = 0)$
$read(z, 0)$
Execution order for process p
$read(y, 0)$
$write(x, 1)$
Time
$x = 0 \wedge y = 0 \wedge z \neq 0$
$write(y, 1)$
$write(z, 0)$
Execution order for process q
$read(k_x, 0)$
$\mathsf{eval}_q^{\mathcal{R}}(k_y = 1)$
$\mathsf{update}_q^{\mathcal{R}}(y, k_y)$
$\mathsf{eval}_q^{\mathcal{R}}(k_x = x)$
$\mathsf{update}_q^{\mathcal{R}}(z, k_x)$
$\mathsf{idle}_q^{\mathcal{R}}$
Program order for process q
$[\![y := 1]\!]_q^{\mathcal{R}}$
$[\![z := x]\!]_q^{\mathcal{R}}$
$[\![\,\mathsf{Idle}\,]\!]_q^{\mathcal{R}}$

Fig. 11. A possible RTSO execution of $C_p \mid\mid C_q$ from Fig. 7

truly concurrent non-conflicting reads and writes to take place. Conflicts are avoided by fractional permissions together with assumptions **HC1** and **HC2**.

4.4 Total Store Order Semantics

The TSO memory model extends RTSO by allowing a process to read values from its own write buffer early without waiting for the pending writes to be committed to memory. Hence, in the TSO memory model, a read to a variable v returns the pending value of v in the write buffer (if a pending write exists) and the value of v from memory (if there are no pending writes to v). It is possible for a buffer to contain multiple pending writes to v, i.e. the same variable occurs more than once in a write buffer; in this case, a read must return the most recent pending write.

We let seq.T denote sequences of type T, assume sequences are indexed from 0 onward, let $\langle a_0, \dots, a_{n-1}\rangle$ denote a sequence with n elements and use '$\frown$' to denote sequence concatenation. To formalise the semantics of a program under TSO, we further extend the state and explicitly include a variable $Buffer_p$ whose value is of type $\mathrm{seq}.(Var \times Val)$ and models the write buffer of process p. Each $Buffer_p$ is a sequence containing pending writes with its new value. Hence, we define two state predicates for a variable v, process p, value k, and state σ. We assume dom.f and ran.f return the domain and range of function f, respectively.

$$\begin{aligned} inBuffer.v.p.\sigma &\mathrel{\widehat{=}} \exists k \bullet (v, k) \in \mathrm{ran}.(\sigma.Buffer_p) \\ bufferVal.v.k.p.\sigma &\mathrel{\widehat{=}} \exists i \bullet \sigma.Buffer_p.i = (v, k) \wedge \\ &\qquad\qquad \forall j \colon \mathrm{dom}.(\sigma.Buffer_p), l \colon Val \bullet j > i \Rightarrow \sigma.Buffer_p.j \neq (v, l) \end{aligned}$$

Hence, $inBuffer.v.p.\sigma$ holds iff there is a pending write to v in the write buffer of p in state σ, and $bufferVal.v.k.p.\sigma$ holds iff the latest value of v in $Buffer_p$ of state σ is k.

We define the set of states apparent to a process p under TSO memory with respect to set of variables W as follows, assuming that the evaluation takes place in stream s within interval $\varDelta$ in the program order.

$$apparent^{\mathcal{T}}_{p,W}.\Delta.s \mathrel{\widehat{=}} \left\{ \sigma\colon State \;\middle|\; \begin{array}{l} \forall v\colon W \bullet \textbf{if } inBuffer.v.p.(s.(\mathsf{glb}.\Delta)) \\ \quad \textbf{then } bufferVal.v.(\sigma.v).p.(s.(\mathsf{glb}.\Delta)) \\ \quad \textbf{else } \exists \gamma\colon apparent^{\mathcal{R}}_{p,W}.\Delta.s \bullet \sigma.v = \gamma.v \end{array} \right\}$$

As with the other memory models, we generate an apparent state by mapping each variable in W to a possible value over the evaluation interval. If $v \in W$ is in p's write buffer, the value of v is taken from the most recent write to v within the write buffer. Otherwise, we return a possible value with respect to RTSO evaluation.

Using $apparent^{\mathcal{T}}_{p,vars.c}$, we define an operator that formalises whether a state predicate holds in some apparent state with respect to an interval Δ and stream s as follows.

$$(\diamondsuit\!\!\!\!\!\raisebox{0.1ex}{\tiny T}_p c).\Delta.s \mathrel{\widehat{=}} \exists \sigma\colon apparent^{\mathcal{T}}_{p,vars.c}.\Delta.s \bullet c.\sigma$$

To complete the program semantics, we instantiate functions $\mathsf{idle}^{\mathcal{T}}_p$, $\mathsf{eval}^{\mathcal{T}}_p$ and $\mathsf{update}^{\mathcal{T}}_p$ for a process p as follows.

$$\begin{aligned} \mathsf{idle}^{\mathcal{T}}_p.V &\mathrel{\widehat{=}} \mathsf{idle}^{\mathcal{R}}_p.V \wedge \mathsf{stable}.Buffer_p \\ \mathsf{eval}^{\mathcal{T}}_p.c &\mathrel{\widehat{=}} \diamondsuit_p c \wedge (\mathsf{idle}_p \wedge \mathsf{wBar}_p).Var \wedge \mathsf{rBar}_p.(Var \backslash vars.c) \wedge \mathsf{stable}.Buffer_p \\ \mathsf{update}^{\mathcal{T}}_p(v,k) &\mathrel{\widehat{=}} \left(\exists buf \bullet \ominus(\overrightarrow{buf = Buffer_p}) \wedge \boxdot(Buffer_p = buf \frown \langle (v,k) \rangle)\right) \wedge \\ &\qquad \mathsf{idle}^{\mathcal{R}}_p.Var \wedge \neg\mathsf{empty} \end{aligned}$$

Command $\mathsf{idle}^{\mathcal{T}}_p$ behaves as $\mathsf{idle}^{\mathcal{R}}_p.V$ and in addition ensures that $Buffer_p$ is not modified. Interval predicate $\mathsf{eval}^{\mathcal{T}}_p.c$ holds iff c holds in some apparent state using TSO evaluation, and in addition, does not modify any variable (including $Buffer_p$), place a write barrier to any variable, or place a read barrier to any variable outside of $vars.c$. Finally, $\mathsf{update}^{\mathcal{T}}_p(v,k)$ adds the pair (v,k) to the end of $Buffer_p$, which is obtained from the state at the end of an immediately preceding interval, and behaves as $\mathsf{idle}^{\mathcal{R}}_p.Var$, i.e., does not modify the value or barrier of any variable. Interval predicate $\mathsf{update}^{\mathcal{T}}_p(v,k)$ also ensures that the interval under consideration is non-empty to guarantee that the buffer is actually updated.

Unlike $\mathcal{I}$, $\mathcal{S}$ and $\mathcal{R}$, local writes in a process p are not visible to other concurrent processes as long as the writes are stored in the buffer of p. To make these local writes globally visible, p must commit any pending writes to shared memory, which is achieved via a *flush* command. Buffers must be flushed in a FIFO order. Note that a flush does not necessarily commit the contents of the entire buffer, and may also not commit any elements from the buffer. Hence, we define an interval predicate commit_p, which commits the first pending write from $Buffer_p$ to memory, then extend the language with a basic command Flush, which for a process p, commits the first k elements of $Buffer_p$, where the value of k is chosen non-deterministically.

$$\begin{aligned} \mathsf{commit}_p &\mathrel{\widehat{=}} \exists buf, v, k \bullet \ominus(\overrightarrow{buf = Buffer_p \wedge (v,k) = buf.0}) \wedge \\ &\qquad [\![\, \mathsf{fin\_Idle} \,]\!]^{\mathcal{T}}_p \ ; \ (\mathsf{update}^{\mathcal{R}}_p(v,k) \wedge \boxdot(Buffer_p = tail.buf)) \\ [\![\, \mathsf{Flush} \,]\!]^{\mathcal{T}}_p &\mathrel{\widehat{=}} \exists k\colon \mathrm{dom}.Buffer_p \cup \{-1\} \bullet \mathsf{commit}^{k+1} \end{aligned}$$

Hence, $\mathsf{commit}^{\mathcal{T}}_p$ instantiates buf to be the value of $Buffer_p$ at the end of the previous interval and sets the pair (v,k) to be the first element of buf. It then performs some finite idling, then the value of p is updated in the same manner as for RTSO and the value of the $Buffer_p$ is set to be $tail.buf$, which is the remaining write buffer excluding the first element of *buf*. Using $\mathsf{update}_p^{\mathcal{R}}(v,k)$ in order to commit elements to memory ensures that the required write barriers to v are placed appropriately. Note that empty implies finite idling, and hence, elements from the buffer may also be committed immediately. The behaviour of $[\![\,\mathsf{Flush}\,]\!]_p^{\mathcal{T}}$ commits 0 or more pending writes (upto the number of elements in the buffer). If the buffer is empty, the only possible behaviour of Flush is $\mathsf{commit}^0 \equiv \mathsf{empty}$.

Processes under TSO may non-deterministically choose to commit contents from their write buffer to memory, therefore, the semantics of a basic command BC allows an arbitrary number of Flush commands after the execution of BC.

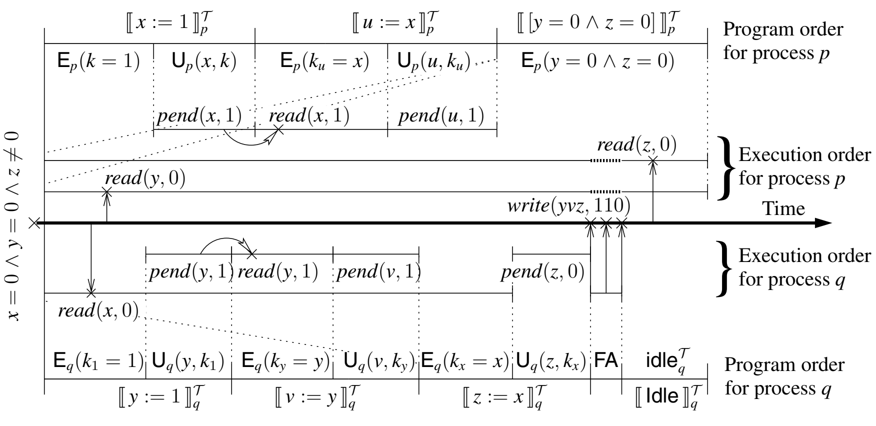
$$[\![\,BC\,]\!]_p^{\mathcal{T}} \mathrel{\widehat{=}} (\!|\,BC\,|\!)_p^{\mathcal{T}} \ ; \ [\![\,\mathsf{Flush}\,]\!]_p^{\mathcal{T}}$$

Note that $[\![\,BC_1\,]\!]_p^{\mathcal{T}} \ ; \ [\![\,BC_2\,]\!]_p^{\mathcal{T}}$ is a possible behaviour of $[\![\,BC_1 \ ; \ BC_2\,]\!]_p^{\mathcal{T}}$, where BC_1 and BC_2 are both basic commands because $[\![\,\mathsf{Flush}\,]\!]_p^{\mathcal{T}}$ may be instantiated to commit^0, which is equivalent to empty and $(g_1 \ ; \ \mathsf{empty} \ ; \ g_2) \equiv (g_1 \ ; \ g_2)$ for any interval predicates g_1 and g_2. That is, a buffer flush may not occur in between two consecutive commands. TSO guarantees that the buffer is eventually flushed. This is incorporated into our semantics by modifying the behaviour of parallel composition so that the entire buffer is flushed when the processes terminate.

$$\begin{aligned}
&[\![\,\mathsf{FlushAll}\,]\!]_P^{\mathcal{T}} \mathrel{\widehat{=}} \textstyle\bigwedge_{p:P} \mathsf{commit}^{\#\mathrm{dom}.\mathit{Buffer}_p} \\
&[\![\,\|_{p:P}\, C_p\,]\!]_P^{\mathcal{T}} \mathrel{\widehat{=}} \textbf{if } P = \varnothing \textbf{ then } \mathit{true} \textbf{ elseif } P = \{p\} \textbf{ then } [\![\,C_p\,]\!]_{\{p\}}^{\mathcal{T}} \\
&\qquad \textbf{else } \exists Q, R, S, T \bullet (Q \cup R = P) \wedge (Q \cap R = \varnothing) \wedge Q \neq \varnothing \wedge R \neq \varnothing \wedge \\
&\qquad\qquad \mathit{term}.S.T \wedge [\![\,(\|_{p:Q}\, C_p) \ ; \ \mathsf{FlushAll} \ ; \ S\,]\!]_Q^{\mathcal{T}} \wedge \\
&\qquad\qquad [\![\,(\|_{p:R}\, C_p) \ ; \ \mathsf{FlushAll} \ ; \ T\,]\!]_R^{\mathcal{T}}
\end{aligned}$$

Example. An example execution under TSO is given in Fig. 12, where the E_p and U_p abbreviate $\mathsf{eval}_p^{\mathcal{T}}$ and $\mathsf{update}_p^{\mathcal{T}}$, respectively, FA denotes execution of a $\mathsf{FlushAll}$ command, and $\mathit{pend}(v,k)$ denotes an enqueuing of a pending write to the buffer of the corresponding process. Like Fig. 11, execution intervals are shown below the program order of process p (above q, respectively). In the depicted execution, each U adds a pending write to the local buffer, and hence, the new value cannot be observed by the other process. Because pending values are read first, execution of $\mathsf{E}_p(k_u = x)$ reads the value of x from the buffer. The effects of $\mathsf{U}_p(x,k)$, $\mathsf{E}_p(k_u = x)$ and $\mathsf{U}_p(u,k_u)$ are local, and hence, do not place any read barriers. This allows the interval of execution of $[y = 0 \wedge z = 0]$ to be extended as depicted. Following a similar behaviour in process q, variables y and x can be read early in processes p and q, enabling $[y = 0 \wedge z = 0]$ in process p to evaluate to *true*. One can also see that $[y = 0 \wedge z = 0]$ can never evaluate to *true* under RTSO memory because the intervals of evaluation cannot be extended beyond preceding evaluation intervals.

Note that the pending writes are eventually are committed to memory, which is only shown in Fig. 12 for process q, but is omitted for process p due to lack of space. The example also shows how memory efficiency has been improved by avoiding a read of x in process p and a read of y in process q. Furthermore, by storing pending writes in a buffer, the processes are able to wait until contention for shared memory has reduced before committing their writes.



$[\![x := 1]\!]_p^{\mathcal{T}}$
$[\![u := x]\!]_p^{\mathcal{T}}$
$[\![[y = 0 \wedge z = 0]]\!]_p^{\mathcal{T}}$
Program order for process p
$\mathsf{E}_p(k = 1)$
$\mathsf{U}_p(x, k)$
$\mathsf{E}_p(k_u = x)$
$\mathsf{U}_p(u, k_u)$
$\mathsf{E}_p(y = 0 \wedge z = 0)$
$pend(x, 1)$
$read(x, 1)$
$pend(u, 1)$
$read(z, 0)$
$read(y, 0)$
Execution order for process p
$write(yvz, 110)$
Time
$x = 0 \wedge y = 0 \wedge z \neq 0$
$pend(y, 1)$
$read(y, 1)$
$pend(v, 1)$
$pend(z, 0)$
Execution order for process q
$read(x, 0)$
$\mathsf{E}_q(k_1 = 1)$
$\mathsf{U}_q(y, k_1)$
$\mathsf{E}_q(k_y = y)$
$\mathsf{U}_q(v, k_y)$
$\mathsf{E}_q(k_x = x)$
$\mathsf{U}_q(z, k_x)$
FA
$\mathsf{idle}_q^{\mathcal{T}}$
Program order for process q
$[\![y := 1]\!]_q^{\mathcal{T}}$
$[\![v := y]\!]_q^{\mathcal{T}}$
$[\![z := x]\!]_q^{\mathcal{T}}$
$[\![\mathsf{Idle}]\!]_q^{\mathcal{T}}$

Fig. 12. A possible TSO execution of $D_p \parallel D_q$ from Fig. 8

5 Conclusions and Future Work

This paper presents a high-level formalisation of a program's behaviour under Total Store Order memory using an interval-based semantics. We enable reasoning about the fine-grained atomicity of expression evaluation that not only captures the inherent non-determinism due to concurrency, but also due to the memory read/write policy of the underlying hardware. Our formalisation is presented at a level of abstraction that avoids compilation to low-level language. Hence, tedious transformations steps (e.g. encoding of additional control-flow due to reorderings) are not necessary, and therefore, compares favourably to the existing low-level formalisations in the literature. The presented semantics is modular in the sense that the underlying memory model is a parameter to the behaviour function. The separation of program specification and language semantics that our framework achieves is beneficial in the sense that it reduces the specification effort.

We aim to use the semantics from this paper to reason about concurrent programs using interval-based rely/guarantee reasoning [8, 9]. In particular, to show that a program modelled by C executed by a set of processes P under memory model $\mathcal{M}$ satisfies a property g (expressed as an interval predicate), one would need to prove a formula of the form $[\![C]\!]_P^{\mathcal{M}} \Rightarrow g$. If C is a parallel composition $\|_{p:P}\, C_p$, one can decompose proofs of $[\![\|_{p:P}\, C_p]\!]_P^{\mathcal{M}} \Rightarrow g$ into proofs $[\![\|_{p:Q}\, C_p]\!]_Q^{\mathcal{M}} \Rightarrow r$ and $[\![\|_{p:R}\, C_p]\!]_R^{\mathcal{M}} \wedge r \Rightarrow g$ where Q and R are disjoint sets such that $Q \cup R = P$ and r is an interval predicate that represents a rely condition [9, 11, 17].

Using our formalisation it is possible to prove relationships between different memory models, e.g., that SC is a special case of RTSO, which in turn is a special case of TSO, but we leave these proofs as future work. Other future work includes mechanisation of the language semantics in a theorem prover, and the development of high-level semantics for other weak memory models such as PSO [7] and transactional memory [14], together with proofs that relate the various semantics.

References

1. Adve, S.V., Gharachorloo, K.: Shared memory consistency models: A tutorial. IEEE Computer 29(12), 66–76 (1996)
2. Alglave, J., Kroening, D., Nimal, V., Tautschnig, M.: Software verification for weak memory via program transformation. CoRR, abs/1207.7264 (2012)
3. Arvind, A., Maessen, J.-W.: Memory model = instruction reordering + store atomicity. SIGARCH Comput. Archit. News 34(2), 29–40 (2006)
4. Atig, M.F., Bouajjani, A., Burckhardt, S., Musuvathi, M.: On the verification problem for weak memory models. In: POPL, pp. 7–18. ACM, New York (2010)
5. Boyland, J.: Checking interference with fractional permissions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 55–72. Springer, Heidelberg (2003)
6. Burckhardt, S., Alur, R., Martin, M.M.K.: Checkfence: Checking consistency of concurrent data types on relaxed memory models. In: PLDI, pp. 12–21 (2007)
7. Inc. CORPORATE SPARC International. The SPARC architecture manual: version 8. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1992)
8. Dongol, B., Derrick, J.: Proving linearisability via coarse-grained abstraction. CoRR, abs/1212.5116 (2012)
9. Dongol, B., Derrick, J., Hayes, I.J.: Fractional permissions and non-deterministic evaluators in interval temporal logic. ECEASST 53 (2012)
10. Dongol, B., Hayes, I.J.: Deriving real-time action systems controllers from multiscale system specifications. In: Gibbons, J., Nogueira, P. (eds.) MPC 2012. LNCS, vol. 7342, pp. 102–131. Springer, Heidelberg (2012)
11. Dongol, B., Hayes, I.J.: Rely/guarantee reasoning for teleo-reactive programs over multiple time bands. In: Derrick, J., Gnesi, S., Latella, D., Treharne, H. (eds.) IFM 2012. LNCS, vol. 7321, pp. 39–53. Springer, Heidelberg (2012)
12. Dongol, B., Hayes, I.J., Meinicke, L., Solin, K.: Towards an algebra for real-time programs. In: Kahl, W., Griffin, T.G. (eds.) RAMICS 2012. LNCS, vol. 7560, pp. 50–65. Springer, Heidelberg (2012)
13. Hayes, I.J., Burns, A., Dongol, B., Jones, C.: Comparing degrees of non-determinism in expression evaluation. The Computer Journal (2013) (accepted January 04, 2013)
14. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: Jay Smith, A. (ed.) ISCA, pp. 289–300. ACM (1993)
15. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (1990)
16. Intel, Santa Clara, CA, USA. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1 (May 2012)
17. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Trans. Prog. Lang. and Syst. 5(4), 596–619 (1983)
18. Jones, C.B., Pierce, K.: Elucidating concurrent algorithms via layers of abstraction and reification. Formal Aspects of Computing 23, 289–306 (2011)
19. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Computers 28(9), 690–691 (1979)
20. Morgan, C.: Programming from Specifications. Prentice-Hall (1990)
21. Moszkowski, B.C.: A complete axiomatization of Interval Temporal Logic with infinite time. In: LICS, pp. 241–252 (2000)
22. AMD64 Architecture Programmer's Manual Volume 2: System Programming (2012), `http://support.amd.com/us/Processor_TechDocs/24593_APM_v2.pdf`
23. Park, S., Dill, D.L.: An executable specification, analyzer and verifier for RMO (relaxed memory order). In: SPAA, pp. 34–41 (1995)
24. Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z., Myreen, M.O.: x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. Commun. ACM 53(7), 89–97 (2010)